

Response-Time Analysis of DAG Tasks Supporting Heterogeneous Computing

Maria A. Serrano

Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain
maria.serranogracia@bsc.es

Eduardo Quiñones

Barcelona Supercomputing Center (BSC)
Barcelona, Spain
eduardo.quinones@bsc.es

ABSTRACT

Hardware platforms are evolving towards parallel and heterogeneous architectures to overcome the increasing necessity of more performance in the real-time domain. Parallel programming models are fundamental to exploit the performance capabilities of these architectures. This paper proposes a novel response time analysis (RTA) for verifying the schedulability of DAG tasks supporting heterogeneous computing. It analyzes the impact of executing part of the DAG in the accelerator device. As a result, the response time upper bound of the system is more precise than the one provided by currently existing RTA targeting homogeneous architectures.

1 INTRODUCTION

Parallel and heterogeneous hardware architectures become mainstream in the embedded domain to cope the increasing performance requirements. These architectures integrate low power general-purpose multi-cores (known as *host*) with dedicated accelerator devices like DSP fabrics, GPUs or FPGAs. Some examples are the NVIDIA Tegra X1[14], TI Keystone II[20] or Xilinx UltraScale[11].

Parallel programming models are fundamental to effectively exploit the huge performance capabilities of these architectures. As an example, OpenMP [15] is increasingly being adopted in architectures targeting embedded systems. As a matter of fact, all parallel architectures presented above support OpenMP in their software development kit. Moreover, OpenMP incorporates a host-centric acceleration model to efficiently offload code and data to devices.

Functional and non-functional verification is fundamental when designing real-time embedded systems. However, the use of parallel programming models like OpenMP involves many challenges to assure that software satisfies both functional and non-functional requirements. This paper addresses the latter, focusing on the worst-case response time analysis of OpenMP programs. It is worth mentioning however that recent works have addressed functional verification of OpenMP programs [16], demonstrating the benefits of using OpenMP in real-time embedded systems.

Regarding timing verification, the sporadic DAG task model [6] analyzes the response time of systems composed of parallel tasks modeled with direct acyclic graphs (DAGs) [4, 9, 12]. Interestingly,

recent works demonstrate that this model resembles the OpenMP tasking model [13, 19, 21, 22]. However, none of the previous works consider the impact that heterogeneous computing has on the non-functional response time analysis verification.

This paper introduces a new response time analysis for the sporadic DAG tasks model supporting parallel and heterogeneous computing. In heterogeneous computing, the workload offloaded into the accelerator device does not cause any interference on the parallel workload executed in the host, and vice versa. Therefore, our analysis takes this into account and first identifies the portion of the DAG that can potentially execute in parallel with the offloaded workload. Then, DAG transformation techniques are used to safely reduce the interference factor, given that the impact of offloading workload is to reduce the interference in the host.

Our results reveal that, compared to existing RTA targeting homogeneous architectures, the response time is significantly reduced when the offloaded computation represents more than 10% of the overall DAG task workload. In fact, our response time is comparable to the one obtained with an ILP solution (only applicable for small DAGs composed up to 100 nodes). Interestingly, our DAG transformation technique allows to improve the average performance of the task up to 23% when considering a host featuring 16 cores.

2 SYSTEM MODEL

Consider a parallel heterogeneous architecture composed of a host processor with m identical cores and a single accelerator device (e.g. a FPGA, GPU or DSP fabric). Moreover, consider a *host-centric acceleration model* in which the host offloads code and data to the accelerator device and collects results.

A parallel real-time task is represented by $\tau = \langle G, T, D \rangle$. $G = (V, E)$ is the DAG modeling its parallel execution. $V = \{v_1, v_2, \dots, v_n, v_{Off}\}$ is the set of nodes. Nodes $v_i \in V$, $1 \leq i \leq n$ represent sequential jobs executed in the host and node v_{Off} represents the workload executed in the accelerator device, named *offloaded node* (there is only one). All nodes in V are characterized by its *worst-case execution time* (WCET) C_i or C_{Off} . $E = V \times V$ is the set of edges representing precedence constraints among pairs of nodes. If $(v_1, v_2) \in E$, then v_1 must complete before v_2 can begin execution (transitive edges do not exist, i.e. if $(v_1, v_2) \in E$ and $(v_2, v_3) \in E$ then $(v_1, v_3) \notin E$). Nodes with no incoming arcs are *sources* of the DAG, while nodes with no outgoing arcs are *sinks*. Without loss of generality, we assume that each DAG has exactly one source v_1 and one sink v_n node. If this is not the case, a dummy source/sink node with zero WCET can be added to the DAG, with edges to/from all the source/sink nodes. Finally, T is the minimum inter-arrival time of τ and D is the constrained relative deadline ($D \leq T$).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5700-5/18/06...\$15.00

<https://doi.org/10.1145/3195970.3196104>

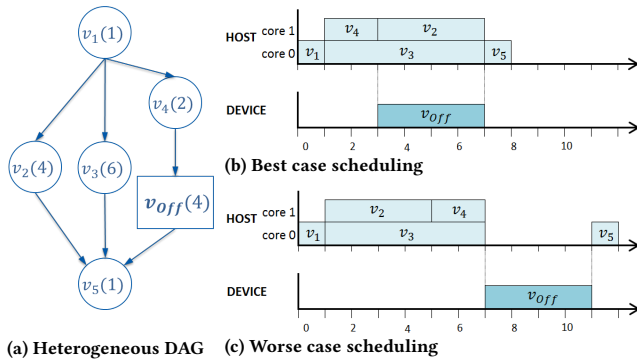


Figure 1: Heterogeneous DAG task scheduling example.

Two interesting DAG properties are $vol(G)$ and $len(G)$: $vol(G) = \sum_{v_j \in V} C_j$ represents the volume of the DAG. In a parallel architecture, the volume denotes the WCET of the task when executing sequentially on a single core in the host and a single accelerator, assuming that core and accelerator cannot execute in parallel. $len(G)$ is the length of the *critical path* of the DAG, i.e. the longest path. It corresponds to the minimum amount of time needed to execute the task on a sufficiently large number of host cores.

Interestingly, our system model resembles the OpenMP parallel programming model [15][21]. OpenMP implements a very advanced and coupled task and host-centric acceleration models. It incorporates easy-to-use data clauses to express data directionality when moving data back and forth to/from the device memories. OpenCL and CUDA provide similar functionality at lower-level. A compiler method to derive an OpenMP-DAG compliant with the OpenMP semantics is proposed in [22]. The OpenMP accelerator model can be easily incorporated into the DAG by distinguishing those nodes executed in the host from those executed in the device.

3 HETEROGENEOUS MODEL

3.1 Starting Point: Homogeneous model

In [19], authors computed a response time upper bound of a DAG task τ running on m homogeneous cores as:

$$R^{hom}(\tau) = len(G) + \frac{1}{m} (vol(G) - len(G)) \quad (1)$$

where $len(G)$ is the length of the critical path of G and $vol(G)$, its volume. The factor $\frac{1}{m} (vol(G) - len(G))$ upper-bounds the *self-interference* i.e., the interference contribution from the task itself to its critical path. In order to verify the schedulability of τ , the result provided by Equation 1 must be compared with τ 's relative deadline D , $R^{hom}(\tau) \leq D$.

3.2 Towards an heterogeneous model

Clearly, *heterogeneous computing reduces the actual interference compared to homogeneous*, as the offloaded node does not occupy resources in the host. However, this interference reduction in the host may not imply a reduction on the response time, as the precedence constraints defined in E may defeat heterogeneous benefits.

In order to illustrate this phenomenon, consider the DAG task τ shown in Figure 1(a) composed of six nodes v_1, \dots, v_5, v_{off} (with WCET shown in parenthesis). The critical path is $\{v_1, v_3, v_5\}$ ($len(G) = 8$). Assuming $m = 2$, the self-interference factor is $\frac{1}{2}(18 - 8) = 5$, resulting in $R^{hom}(\tau) = 13$. Since v_{off} does not

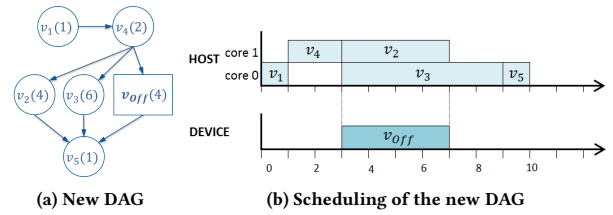


Figure 2: Transformation of the DAG task in Figure 1.

execute in the host, one might subtract its contribution to the self-interference factor, see Figure 1(b), resulting in $R^{hom}(\tau) = 11$. However, the reduction in the self-interference factor does not guarantee a trustworthy response time upper bound because v_{off} may not necessarily execute in parallel with the nodes running in the host. See Figure 1(c) in which all cores in the host remain idle while v_{off} is running. In this case, the response time is 12, which is higher than the reduced $R^{hom}(\tau)$ computed above, 11.

Overall, the DAG portion that *potentially executes* in parallel with the offloaded node (and so reducing the interference) is *not guaranteed to actually execute* in parallel with it.

3.3 Safe self-interference reduction

In order to safely reduce the self-interference factor, it is first necessary to guarantee that there is enough workload to be executed in the host in parallel with v_{off} . To do so, we propose an algorithm that: (1) identifies the sub-DAG that may potentially execute in parallel with v_{off} , named $G^{Par} = (V^{Par}, E^{Par})$, and (2) adds a synchronization point to guarantee that G^{Par} and v_{off} actually execute in parallel.

Figure 2(a) shows the proposed transformation of the DAG presented in Figure 1(a). Hence, by inserting a synchronization point between nodes v_4 and v_2, v_3 , it is guaranteed that v_{off} and $\{v_2, v_3\}$ execute in parallel. Figure 2(b) shows the scheduling of the transformed DAG. Synchronization forces v_1 and v_4 to be scheduled first, avoiding the scheduling scenario shown in Figure 1(c).

Clearly, this strategy may impact on the average performance of the tasks because: (1) the critical path can potentially enlarge (e.g., the length of the transformed DAG in Figure 2(a) is 10 instead of 8 in the original DAG) and (2) the potential parallelism is reduced due to the synchronization point (e.g., in Figure 2(a), v_4 can not longer be executed in parallel with v_2 and v_3). Interestingly, our experiments demonstrate the opposite effect when the offloaded workload is large enough (see Section 5.2). The reason is that, ensuring the parallel executing of G^{Par} and v_{off} avoids scheduling scenarios in which the offloaded node is running while the host processor remains idle, as shown in Figure 1(c).

Overall, this strategy allows to derive a RTA for heterogeneous architectures. It will be presented in Section 4 but first, we introduce the algorithm to transform the DAG.

3.4 DAG Transformation Algorithm

Algorithm 1 generates a new DAG in which the parallel execution of v_{off} and G^{Par} is guaranteed. To do so, given a DAG $G = (V, E)$, it first generates the transformed DAG $G' = (V', E')$ which includes a new *synchronization node* v_{sync} ($C_{sync} = 0$). Then, it identifies the sub-DAG $G^{Par} = (V^{Par}, E^{Par})$ which includes all the nodes that

can potentially execute in parallel with v_{Off} . v_{sync} is introduced just before v_{Off} and G^{Par} to simultaneously begin execution.

Consider the example shown in Figure 3 in order to facilitate the algorithm explanation. Figure 3(a) shows the original DAG G , in which the synchronization point to be included is represented with a dashed red line. Figure 3(b) shows the resultant DAG G' , including the new synchronization node v_{sync} represented as a red square node, and G^{Par} .

3.4.1 Initialization. First $Pred(v_{Off})$, the set of nodes from which v_{Off} can be reached, and $Succ(v_{Off})$, the set of nodes reachable from v_{Off} , are computed (line 1). Then, the algorithm initializes V' , which includes all the original nodes in V plus the synchronization node v_{sync} , and E' , which includes all the edges in E . A local variable *directPred* is used to store v_{Off} 's direct predecessors¹.

3.4.2 Loop over v_{Off} 's direct predecessors. The transformation starts with a loop (line 3) which iterates over v_{Off} 's direct predecessors, v_i , and (1) adds v_i to *directPred*, (2) adds an edge from v_i to the extra synchronization node v_{sync} and (3) removes (v_i, v_{Off}) edge. In Figures 3(a) and 3(b) this loop operates over nodes v_8 and v_9 to remove their edge with v_{Off} and to add new edges with the new node v_{sync} (green edges). The nested loop in line 6 updates the edges between v_i and v_i 's successors (parallel nodes to v_{Off}) since they are now v_{sync} 's successors. In Figures 3(a) and 3(b) this loop removes (v_8, v_{11}) and adds (v_{sync}, v_{11}) , black edges.

In line 9 a new edge between the extra synchronization node v_{sync} and the offloaded node v_{Off} is added. This corresponds to the yellow edge (v_{sync}, v_{Off}) in Figure 3(b).

3.4.3 Loop over other v_{Off} 's predecessors. The second part of the algorithm, starting in line 10, iterates over all the nodes v_i from which v_{Off} can be reached, except its direct predecessors. Then, a nested loop is used to check if v_i 's successors, named v_j , are parallel to v_{Off} (line 12). If this is the case, then v_j is now a v_{sync} 's successor instead of a v_i 's successor (line 13). Notice that, since transitive edges do not exist, it is not required to check if v_j is in $Succ(v_{Off})$ to determine if v_j is parallel to v_{Off} . In Figures 3(a) and 3(b), these nested loops are used to remove pink edges (v_1, v_2) and (v_3, v_7) and to add pink edges (v_{sync}, v_2) and (v_{sync}, v_7) .

3.4.4 Creating G^{Par} . Finally, the parallel sub-DAG G^{Par} is created, containing all the parallel nodes to v_{Off} (line 14) and all the corresponding edges involving these nodes (line 17). In Figure 3(b) G^{Par} is surrounded by a dashed blue line.

4 RESPONSE TIME ANALYSIS FOR HETEROGENEOUS DAG TASKS

In this section we extend the RTA presented in Equation 1 to support heterogeneous computation. Our analysis is based on the transformed DAG task τ' in which G^{Par} and v_{Off} are guaranteed to execute in parallel. This allows a reduction of the self-interference factor, being the new response time upper bound more accurate than R^{hom} .

Figure 4 shows the generic structure of a transformed DAG task τ' . Given this structure, since there is a synchronization node v_{sync} ,

¹If $(v_i, v_j) \in E$ then v_i is a direct predecessor of v_j .

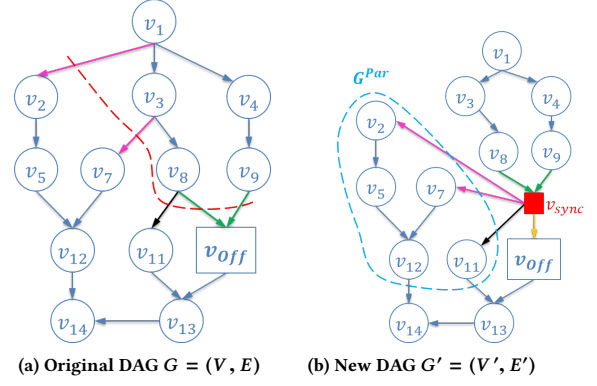


Figure 3: Heterogeneous DAG task transformation $\tau \Rightarrow \tau'$.

Algorithm 1 Transform DAG $\tau \Rightarrow \tau'$

Input: $G = (V, E)$
Output: (1) $G' = (V', E')$; (2) $G^{Par} = (V^{Par}, E^{Par})$

- 1: Compute $Pred(v_{Off})$ and $Succ(v_{Off})$;
- 2: $V' = V \cup \{v_{sync}\}$; $E' = E$; $directPred = \emptyset$;
- 3: **for each** $(v_i, v_{Off}) \in E'$ **do**
- 4: $directPred = directPred \cup \{v_i\}$
- 5: $E' = E' \cup \{(v_i, v_{sync})\} \setminus \{(v_i, v_{Off})\}$
- 6: **for each** $(v_i, v_j) \in E'$ **do**
- 7: **if** $v_j \neq v_{sync}$ **then**
- 8: $E' = E' \cup \{(v_{sync}, v_j)\} \setminus \{(v_i, v_j)\}$
- 9: $E' = E' \cup \{(v_{sync}, v_{Off})\}$
- 10: **for each** $v_i \in \{Pred(v_{Off}) \setminus directPred\}$ **do**
- 11: **for each** $(v_i, v_j) \in E'$ **do**
- 12: **if** $v_j \notin Pred(v_{Off})$ **then**
- 13: $E' = E' \cup \{(v_{sync}, v_j)\} \setminus \{(v_i, v_j)\}$
- 14: $V^{Par} = V \setminus Pred(v_{Off}) \setminus Succ(v_{Off})$
- 15: **for each** $(v_i, v_j) \in E$ **do**
- 16: **if** $v_i, v_j \in V^{Par}$ **then**
- 17: $E^{Par} = E^{Par} \cup \{(v_i, v_j)\}$

the relationship between the G^{Par} and v_{Off} can be classified as follows: either (1) the response time upper bound of G^{Par} , denoted as $R^{hom}(G^{Par})$ ², is bigger or equal than the offloaded workload C_{Off} (see Figure 5(a)); or (2) C_{Off} is bigger than $R^{hom}(G^{Par})$ (see Figure 5(b)). From this relationship, the following theorem considers three possible execution scenarios to propose a new response time analysis supporting heterogeneous computing:

THEOREM 1. Consider an heterogeneous DAG task τ' whose general structure is shown in Figure 4. Depending on the execution scenario, its response time upper bound is computed as follows:

- Scenario 1. v_{Off} does not belong to the critical path.

$$R^{het}(\tau') = len(G') + \frac{1}{m} (vol(G') - len(G') - C_{Off}) \quad (2)$$

- Scenario 2.1. v_{Off} belongs to the critical path and $C_{Off} \geq R^{hom}(G^{Par})$.

$$R^{het}(\tau') = len(G') + \frac{1}{m} (vol(G') - len(G') - vol(G^{Par})) \quad (3)$$

² $R^{hom}(G^{Par})$ is computed with Equation 1. Notice that, for simplicity, the input is a DAG structure G^{Par} instead of a task τ .

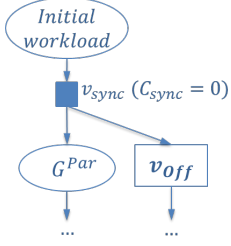


Figure 4: Generic heterogeneous DAG.

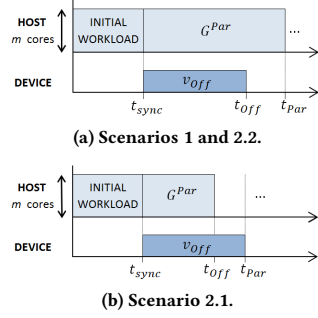


Figure 5: Scheduling possibilities of the generic DAG task in Figure 4.

- *Scenario 2.2.* v_{off} belongs to the critical path and $C_{off} \leq R^{hom}(G^{Par})$.

$$R^{het}(\tau') = len(G') - C_{off} + len(G^{Par}) + \frac{1}{m} (vol(G') - len(G') - len(G^{Par})) \quad (4)$$

PROOF. The transformed DAG in Figure 4 includes a synchronization node v_{sync} ($C_{sync} = 0$) which guarantees that G^{Par} and v_{off} start their execution at the same time (t_{sync} in Figures 5(a)(b)).

In case of *Scenario 1* (represented in Figure 5(a)), since v_{off} does not belong to the critical path, there exists at least one path in G^{Par} whose length is greater than C_{off} , i.e. $len(G^{Par}) > C_{off}$. Therefore, $R^{hom}(G^{Par}) = len(G^{Par}) + \frac{1}{m} (vol(G^{Par}) - len(G^{Par}))$ must be greater than C_{off} , and so $t_{par} > t_{off}$ from Figure 5(a) is always accomplished. As a consequence, C_{off} does not generate interference that may increase the response time of τ' and it can be safely subtracted from the self-interference factor, as done in Equation 2.

In case of *Scenarios 2.1 and 2.2*, since v_{off} belongs to the critical path, none of the nodes in G^{Par} belong to it and so they contribute to the self-interference factor.

In the former scenario (represented in Figure 5(b)), C_{off} is greater (or equal) than $R^{hom}(G^{Par})$, then $t_{par} \leq t_{off}$ in Figure 5(b) and so G^{Par} cannot generate interference that may increase the response time of τ' . Hence, its complete workload $vol(G^{Par})$ can be safely subtracted from the self-interference factor, as done in Equation 3.

In the latter scenario (represented in Figure 5(a)), C_{off} is smaller (or equal) than $R^{hom}(G^{Par})$ (see $t_{off} \leq t_{par}$ in Figure 5(a)). Therefore, even though v_{off} belongs to the critical path, it does not determine the response time of τ' but G^{Par} does instead. In this case, we can safely replace C_{off} by $R^{hom}(G^{Par})$ in the critical path. Since the contribution of G^{Par} is also considered in the self-interference factor, $vol(G^{Par})$ can be subtracted from it, in order not to count twice for it. By replacing the mentioned terms and subtracting $vol(G^{Par})$ we obtain: $R^{het}(\tau') = len(G') - C_{off} + R^G(G^{Par}) + \frac{1}{m} (vol(G') - len(G') - vol(G^{Par})) = len(G') - C_{off} + len(G^{Par}) + \frac{1}{m} (vol(G^{Par}) - len(G^{Par})) + \frac{1}{m} (vol(G') - len(G') - vol(G^{Par}))$. By simplifying the terms, Equation 4 follows. \square

It is important to remark that scenarios 2.1 and 2.2 are equivalent when $C_{off} = R^{hom}(G^{Par})$. Hence, if starting from Equation 4

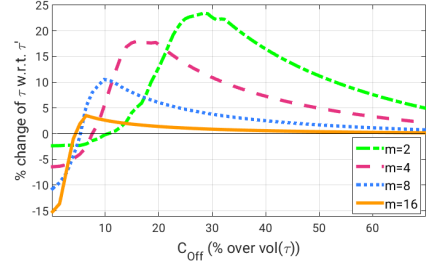


Figure 6: Percentage change of the average execution time of τ w.r.t. τ' when $n \in [100, 250]$.

we replace C_{off} by $R^{hom}(G^{Par}) = len(G^{Par}) + \frac{1}{m} (vol(G^{Par}) - len(G^{Par}))$, we rapidly reach Equation 3.

5 EXPERIMENTAL RESULTS

This section evaluates our response time analysis supporting heterogeneous computing based on randomly generated DAG tasks [12, 18]. In order to evaluate the accuracy of our response time analysis, we implemented an ILP formulation (based on [13]) that computes the minimum time interval needed to execute a given heterogeneous DAG task on m cores and one accelerator device. It provides a node to core mapping so that the heterogeneous DAG task makespan is minimized.

All the algorithms and experiments have been implemented in MATLAB® and the ILP formulation has been coded and solved with the IBM ILOG CPLEX Optimization Studio [10].

5.1 Experimental setup

All experiments consider an heterogeneous architecture composed of a host processor with 2, 4, 8 and 16 cores and one single accelerator device. Random DAG tasks are generated by recursively expanding nodes either to terminal nodes or parallel sub-DAGs, until a maximum recursion depth max_{depth} is reached. max_{depth} also determines the longest possible path of the DAG. The probabilities of generating a parallel sub-DAG or a terminal node are p_{par} and $1 - p_{par}$, respectively. Moreover, the maximum number of branches for any parallel sub-DAG is n_{par} and the minimum and maximum number of nodes of each DAG are n_{min} and n_{max} , respectively. The WCET of each node, except v_{off} , is uniformly selected as a positive integer in the interval $[C_{min}, C_{max}] = [1, 100]$. Once a DAG is generated, we randomly select v_{off} among all the nodes. C_{off} is assigned with the interval $[1, C_{off}^{MAX}]$, where C_{off}^{MAX} represents a percentage (up to 60%) of DAG's volume.

For each experiment, we generate 100 DAGs for each target value of C_{off} . Moreover, we used $p_{par} = 0.5$ and two types of DAG tasks: (1) Small tasks, with $n \leq 100$, $n_{par} = 6$ and $max_{depth} = 3$ (longest path equals 7), used for the ILP solution not capable of dealing with larger tasks and (2) Large tasks with $n \in [100, 400]$, $n_{par} = 8$ and $max_{depth} = 5$ (longest path equals 11).

5.2 Impact of the DAG Transformation

This section evaluates the impact that the extra synchronization point v_{sync} has in the task's performance. To do so, we simulate the execution of the original and transformed DAG tasks (τ and τ' , respectively), assuming the work-conserving *breadth-first scheduler*

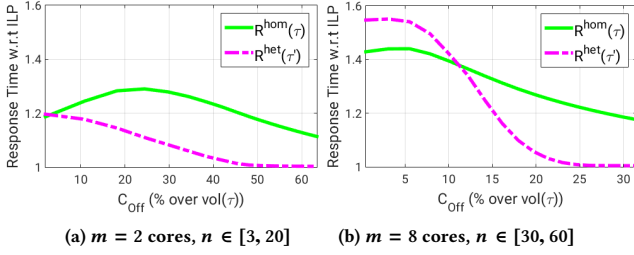


Figure 7: Increment of $R^{het}(\tau')$ and $R^{hom}(\tau)$ w.r.t. the minimum makespan of τ . Notice that x-axes are different.

implemented in GOMP, the OpenMP implementation in the GNU Compiler Collection (GCC) [1].

Figure 6 shows the percentage change³ of the average execution time of τ with respect to τ' , when increasing the offloaded workload C_{Off} with respect to τ 's volume from 1% to 70%. The experiment considers $m = 2, 4, 8$ and 16 cores and a number of nodes $n \in [100, 250]$ (similar trends have been observed when $n \in [250, 400]$).

v_{sync} has a negative impact on the average performance of τ' , compared to τ , when C_{Off} represents a small portion of DAG's volume (less than 11%, 8%, 6% and 4.5% for $m = 2, 4, 8$ and 16, respectively). The reason is that an extra synchronization point limits the parallelism. This negative impact increases as the number of cores increases (and so more parallelism can be potentially exploited). When C_{Off} represents 1% of the DAG's volume, τ is 3% faster than τ' for $m = 2$, and 15% faster for $m = 16$.

Surprisingly, when C_{Off} increases the trend is inverted; τ results 24% slower than τ' for $m = 2$ when C_{Off} represents the 28% of DAG's volume, and 4% slower for $m = 16$ when C_{Off} represents the 8%. The reason is that v_{sync} guarantees that the host processor is not idle while executing v_{Off} (see Figure 1(c)). The performance benefits of v_{sync} decreases as m increases because the self-interference factor has less impact as the number of cores increases (see Theorem 1).

Finally, it is worth noting that for higher values of C_{Off} , the difference between τ and τ' performance seems to decrease. However, the absolute difference remains constant. As C_{Off} increases it becomes the dominant factor in τ and τ' execution times and so both equally increase as well. The trend of the percentage of an absolute difference with respect to an increasing time is to decrease.

5.3 Accuracy of the response time analysis

This section analyses the accuracy of R^{het} (Equations 2, 3, 4) and R^{hom} (Equation 1) with respect to the minimum makespan of a DAG task (ILP solver). Given the ILP complexity, we only consider small tasks for which the ILP solver is able to provide an optimal solution in less than 12 hours.

Figure 7 shows the increment of the response time upper bound provided by $R^{hom}(\tau)$ (Equation 1) and $R^{het}(\tau')$ (Equations 2 to 4) with respect to the minimum makespan of τ computed by the ILP solver, when varying C_{Off} with respect to τ 's volume. We evaluated 2, 4, 8 and 16 cores but for the sake of space, only results for $m = 2$ and 8 cores are shown (Figure 7(a) and (b), respectively).

When C_{Off} represents less than 2% of $vol(\tau)$, $R^{het}(\tau')$ is 19% and 54% higher than the minimum makespan for $m = 2$ and 8,

³The percentage change computes the relative change of two values from the same variable; in our case the average execution time.

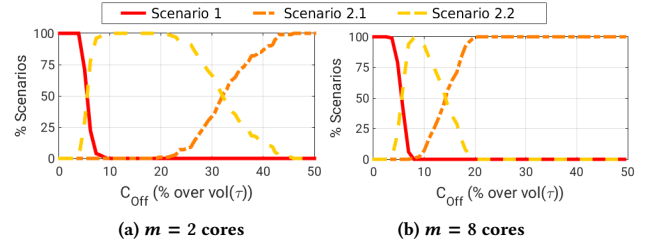


Figure 8: Percentage of scenarios occurrence, $n \in [100, 250]$.

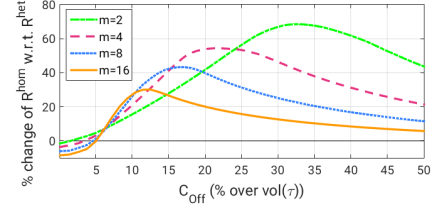


Figure 9: Percentage change of $R^{hom}(\tau)$ w.r.t. $R^{het}(\tau')$, $n \in [100, 250]$.

respectively. For $m = 4$ and 16, $R^{het}(\tau')$ is 40% and 57% higher, respectively (not shown in the figure). This pessimism however decreases as C_{Off} increases, being less than 1% when C_{Off} represents 48.1%, 42.7%, 24.5% and 15% of $vol(\tau)$, for $m = 2, 4, 8$ and 16, respectively. The reason is that C_{Off} becomes the dominant factor of $R^{het}(\tau')$ and so G^{Par} is not relevant any more (see Figure 5(b)).

$R^{hom}(\tau)$ provides more accurate results than $R^{het}(\tau')$ when C_{Off} represents less than 3.1% and 11.2% of $vol(\tau)$ for $m = 2$ and 8, respectively. The reason of this trend, as also shown in Section 5.2, is that v_{sync} impacts negatively on both, average and upper bound response time. For $m = 4$ and 16, $R^{hom}(\tau)$ provides better results when C_{Off} represents less than 12.2% and 8.7%, respectively (not shown in the figure). This trend however is inverted when C_{Off} increases, and so $R^{het}(\tau')$ provides more accurate results than $R^{hom}(\tau)$; e.g. when $R^{het}(\tau')$ provides a response time only 1% higher than the minimum makespan, $R^{het}(\tau')$ is up to 20% higher.

5.4 Homogeneous vs. Heterogeneous

This section evaluates our response time analysis $R^{het}(\tau')$, compared with $R^{hom}(\tau)$. All figures consider randomly generated DAGs with $n \in [100, 250]$ (similar trends observed when $n \in [250, 400]$).

In order to better understand the benefits brought by $R^{het}(\tau')$, it is important first to understand the execution scenarios presented in Theorem 1. For the randomly generated tasks, Figure 8 shows the occurrence percentage of the scenarios described in Section 4, when varying the percentage of C_{Off} over $vol(\tau)$ from 0.12% to 50%. We evaluated heterogeneous architectures featuring host processors with $m = 2, 4, 8$ and 16 cores but, for the sake of space, only results for (a) $m = 2$ and (b) $m = 8$ are shown.

Scenario 1 is the dominant one when the percentage of C_{Off} over $vol(\tau)$ is less than 8%. This scenario corresponds to the case in which v_{Off} does not belong to the critical path and therefore, is independent of m . From that point on, scenario 2.2 becomes more relevant as v_{Off} belongs to the critical path but C_{Off} is still smaller than the response time of G^{Par} . When C_{Off} becomes higher than $R^{hom}(G^{Par})$, occurrences of scenario 2.1 increase. As

m increases, occurrences of scenario 2.1 start to increase earlier because higher parallelism can be exploited in the host, and so $R^{hom}(G^{Par})$ becomes smaller.

Interestingly, intersection of scenarios 2.1 and 2.2, i.e. when $C_{Off} = R^{hom}(G^{Par})$ (Equations 3 and 4 are equivalent), results in maximum benefit of R^{het} with respect to R^{hom} (see below). This occurs when C_{Off} is 32%, 20%, 14% and 10% over $vol(\tau)$ for $m = 2, 4, 8$ and 16, respectively. The reason is that utilization of both host and device is maximized, i.e. there are less idle times.

Figure 9 shows the percentage change of $R^{hom}(\tau)$ with respect to $R^{het}(\tau')$, considering a host processor with $m = 2, 4, 8$ and 16 cores and varying C_{Off} with respect to $vol(\tau)$ from 0.12% to 50%. In general, our response time analysis $R^{het}(\tau')$ improves over $R^{hom}(\tau)$ when considering heterogeneous computation. This improvement increases as C_{Off} increases due to self-interference factor reduction. R^{hom} only outperforms R^{het} for small values of C_{Off} due to the negative impact of the synchronization point. Concretely, this occurs when C_{Off} represents less than 1.6%, 3.4%, 4.6% and 5% over $vol(\tau)$ for $m = 2, 4, 8$ and 16, respectively.

As pointer above, the maximum benefit of $R^{het}(\tau')$ with respect to $R^{hom}(\tau)$ is reached when $C_{Off} = R^{hom}(G^{Par})$. In this case, $R^{hom}(\tau)$ is 70%, 55%, 40% and 30% higher than $R^{het}(\tau')$ for $m = 2, 4, 8$ and 16, respectively. Notice that as m increases, the benefit of $R^{het}(\tau')$ is smaller because the self-interference factor is divided by m (Equations 2 to 4).

Results presented in Figure 9 correspond to an average response time upper bound over all generated DAG tasks. However, the maximum observed difference between $R^{hom}(\tau)$ and $R^{het}(\tau')$ is 95.0%, 82.5%, 65.3% and 47.7% for $m = 2, 4, 8$ and 16, respectively.

6 RELATED WORK

Parallel task models are increasingly being used in the real-time domain. A response-time analysis is presented in [2] for fork/join tasks under partitioned fixed-priority scheduling. The sporadic DAG model [17][6] used in this work has been also considered with conditional nodes [12][5], global [3][18], partitioned [9] or federated [4] scheduling approaches. In [21] authors study the similarities of the DAG model and parallel programming models such as OpenMP. A dynamic schedulability test is provided in [19] and static scheduling heuristics are presented in [13].

Regarding heterogeneous architectures, real-time tasks have been traditionally modeled as self-suspending task. Most of the published work consider that tasks are scheduled on a uniprocessor platform and utilizes a device to accelerate part of the execution. Unfortunately, it has been shown that many previous works concerning the analysis of self-suspending tasks are flawed. Refer to [8] for a complete review of self-suspending tasks theory and an explanation of the existing misconceptions. Finally, in [7] authors design a framework to support real-time systems on FPGAs and provide a response time analysis to verify the schedulability of a set of tasks with software parts and hardware accelerated functions.

7 CONCLUSIONS

This paper presents a novel response time analysis for verifying the schedulability of DAG tasks supporting heterogeneous computing. To do so, we first identify the portion of the DAG running

in the host, G^{Par} , that can potentially execute in parallel with the workload offloaded to the device, v_{Off} . Secondly, we propose a DAG transformation to guarantee the parallel execution of G^{Par} and v_{Off} . We build our response time analysis upon this transformation. Interestingly, besides the timing guarantees provided, this DAG transformation also results in higher average performance when the offloaded workload represents more than 10% of DAG's volume. The reason is that the scenario in which the host processor is idle waiting for the device to finish is avoided. Our response time analysis significantly outperforms the homogeneous one by 70%, 55%, 40% and 30%, for 2, 4, 8 and 16 cores in the host processor, respectively. Moreover, for small DAG tasks (3-100 nodes), our response time upper bound is comparable to the minimum makespan derived with an ILP solution. In the future, we intend to improve the analysis by considering (i) more tasks assigned to the accelerator device, and (ii) more devices in the heterogeneous architecture.

ACKNOWLEDGMENTS

This work is supported by the Spanish Ministry of Science and Innovation under contract TIN2015-65316-P.

REFERENCES

- [1] GOMP project. URL: <https://gcc.gnu.org/projects/gomp/>.
- [2] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Döbel, and H. Härtig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *ECRTS*, 2013.
- [3] S. Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *ECRTS*, 2014.
- [4] S. Baruah. The federated scheduling of systems of mixed-criticality sporadic DAG tasks. In *RTSS*, 2016.
- [5] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *ECRTS*, 2015.
- [6] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. A generalized parallel task model for recurrent real-time processes. In *RTSS*, 2012.
- [7] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo. A framework for supporting real-time applications on dynamic reconfigurable FPGAs. In *RTSS*, 2016.
- [8] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, et al. Many suspensions many problems: A review of self-suspending tasks in real-time systems. *Tech. Rep. 854 (2nd ver.)*, March 2017.
- [9] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *SIES*, 2016.
- [10] IBM ILOG Cplex Optimization studio. URL: <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer>.
- [11] S. Leibson and N. Mehta. Xilinx ultrascale: The next-generation architecture for your next-generation architecture. *Xilinx White Paper WP435*, 2013.
- [12] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *ECRTS*, 2015.
- [13] A. Melani, M. A. Serrano, M. Bertogna, I. Cerutti, E. Quinones, and G. Buttazzo. A static scheduling approach to enable safety-critical OpenMP applications. In *ASP-DAC*, 2017.
- [14] NVIDIA Tegra. K1: A new era in mobile computing. *NVIDIA White Paper*, 2014.
- [15] OpenMP Architecture Review Board. OpenMP 4.5 Complete Specification. URL: <http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>, 2015.
- [16] S. Royuela, M. A. Serrano, A. Duran, X. Martorell, and E. Quinones. A functional safety openmp for critical real-time embedded systems. In *IWOMP*, 2017.
- [17] A. Saifullah, D. Ferry, C. Lu, and C. Gill. Real-time scheduling of parallel tasks under a general DAG model. *Tech. Rep.*, 2012.
- [18] M. A. Serrano, A. Melani, M. Bertogna, and E. Quinones. Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In *DATE*, 2016.
- [19] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quinones. Timing characterization of OpenMP4 tasking model. In *CASES*, 2015.
- [20] Texas Instruments. *66AK2Hxx Multicore Keystone II System-on-Chip (SoC)*. November 2012.
- [21] R. Vargas, E. Quinones, and A. Marongiu. OpenMP and timing predictability: a possible union? In *DATE*, 2015.
- [22] R. E. Vargas, S. Royuela, M. A. Serrano, X. Martorell, and E. Quinones. A lightweight OpenMP4 run-time for embedded systems. In *ASP-DAC*, 2016.